

22 | 知识串讲（下）：带你开发一个书店应用

2020-06-25 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 11:58 大小 10.97M



你好，我是 Chrono。

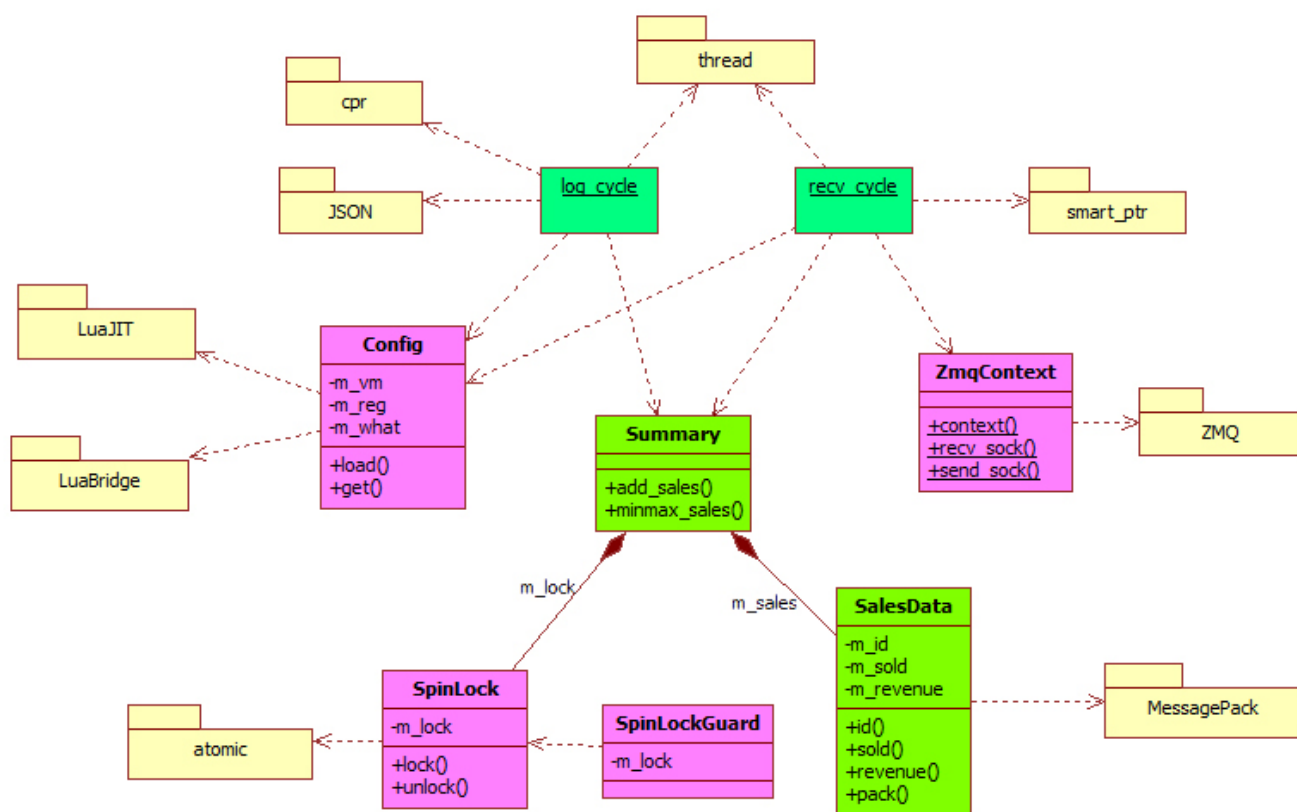
在上节课里，我给出了一个书店程序的例子，讲了项目设计、类图和自旋锁、Lua 配置文件解析等工具类，搭建出了应用的底层基础。

今天，我接着讲剩下的主要业务逻辑部分，也就是数据的表示与统计，还有数据的接收和发送主循环，最终开发出完整的应用程序。

这里我再贴一下项目的 UML 图，希望给你提个醒。借助图形，我们往往能够更好地梳理程序的总体结构。



图中间标注为绿色的两个类 SalesData、Summary 和两个 lambda 表达式 recv_cycle、log_cycle 是今天这节课的主要内容，实现了书店程序的核心业务逻辑，所以需要重点关注它。



数据定义

首先，我们来看一下怎么表示书本的销售记录。这里用的是 SalesData 类，它是书店程序数据统计的基础。

如果是实际的项目，SalesData 会很复杂，因为一本书的相关信息有很多。但是，我们的这个例子只是演示，所以就简化了一些，基本的成员只有三个：ID 号、销售册数和销售金额。

上节课，在讲自旋锁、配置文件等类时，我只是重点说了说代码内部逻辑，没有完整地细说，到底该怎么应用前面讲过的那些 C++ 编码准则。

那么，这次在定义 SalesData 类的时候，我就集中归纳一下。这些都是我写 C++ 代码时的“惯用法”，你也可以在自己的代码里应用它们，让代码更可读可维护：

适当使用空行分隔代码里的逻辑段落；

类名使用 CamelCase，函数和变量用 snake_case，成员变量加 “m_” 前缀；

在编译阶段使用静态断言，保证整数、浮点数的精度；

使用 final 终结类继承体系，不允许别人产生子类；

使用 default 显式定义拷贝构造、拷贝赋值、转移构造、转移赋值等重要函数；

使用委托构造来编写多个不同形式的构造函数；

成员变量在声明时直接初始化；

using 定义类型别名；

使用 const 来修饰常函数；

使用 noexcept 标记不抛出异常，优化函数。

列的点比较多，你可以对照着源码来进行理解：

```
1 class SalesData final // final禁止继承
2 {
3 public:
4     using this_type = SalesData; // 自己的类型别名
5 public:
6     using string_type = std::string; // 外部的类型别名
7     using string_view_type = const std::string&;
8     using uint_type = unsigned int;
9     using currency_type = double;
10
11     STATIC_ASSERT(sizeof(uint_type) >= 4); // 静态断言
12     STATIC_ASSERT(sizeof(currency_type) >= 4);
13 public:
14     SalesData(string_view_type id, uint_type s, currency_type r) noexcept
15         : m_id(id), m_sold(s), m_revenue(r)
16     {}
17
18     SalesData(string_view_type id) noexcept // 委托构造
19         : SalesData(id, 0, 0)
20     {}
21 public:
22     SalesData() = default; // 显式default
23     ~SalesData() = default;
24
25     SalesData(const this_type&) = default;
26     SalesData& operator=(const this_type&) = default;
```

 复制代码

```

27     SalesData(this_type&& s) = default; // 显式转移构造
28     SalesData& operator=(this_type&& s) = default;
29 private:
30     string_type m_id          = "";          // 成员变量初始化
31     uint_type   m_sold        = 0;
32     uint_type   m_revenue     = 0;
33 public:
34     void inc_sold(uint_type s) noexcept      // 不抛出异常
35     {
36         m_sold += s;
37     }
38 public:
39     string_view_type id() const noexcept    // 常函数, 不抛出异常
40     {
41         return m_id;
42     }
43
44     uint_type sold() const noexcept         // 常函数, 不抛出异常
45     {
46         return m_sold;
47     }
48 };
49

```

需要注意的是，代码里显式声明了转移构造和转移赋值函数，这样，在放入容器的时候就避免了拷贝，能提高运行效率。

序列化

SalesData 作为销售记录，需要在网络上传输，所以需要序列化和反序列化。

这里我选择的是 MessagePack ([@第 15 讲](#))，我看重的是它小巧轻便的特性，而且用起来也很容易，只要在类定义里添加一个宏，就可以实现序列化：

```

1 public:
2     MSGPACK_DEFINE(m_id, m_sold, m_revenue); // 实现MessagePack序列化功能

```

 复制代码

为了方便使用，还可以为 SalesData 增加一个专门序列化的成员函数 pack()：

```

1 public:

```

 复制代码

```

2   msgpack::sbuffer pack() const           // 成员函数序列化
3   {
4       msgpack::sbuffer sbuf;
5       msgpack::pack(sbuf, *this);
6
7       return sbuf;
8

```

不过你要注意，写这个函数的同时也给 SalesData 类增加了点复杂度，在一定程度上违反了单一职责原则和接口隔离原则。

如果你在今后的实际项目中遇到类似的问题，就要权衡后再做决策，确认引入新功能带来的好处大于它增加的复杂度，尽量抵制扩充接口的诱惑，否则很容易写出“巨无霸”类。

数据存储与统计

有了销售记录之后，我们就可以定义用于数据存储和统计的 Summary 类了。

Summary 类依然要遵循刚才的那些基本准则。从 UML 类图里可以看到，它关联了好几个类，所以类型别名对于它来说就特别重要，不仅可以简化代码，也方便后续的维护，你可要仔细看一下源码：

```

1  class Summary final                       // final禁止继承
2  {
3  public:
4      using this_type = Summary;           // 自己的类型别名
5  public:
6      using sales_type      = SalesData;    // 外部的类型别名
7      using lock_type       = SpinLock;
8      using lock_guard_type = SpinLockGuard;
9
10     using string_type      = std::string;
11     using map_type          =              // 容器类型定义
12         std::map<string_type, sales_type>;
13     using minmax_sales_type =
14         std::pair<string_type, string_type>;
15 public:
16     Summary() = default;                  // 显式default
17     ~Summary() = default;
18
19     Summary(const this_type&) = delete;   // 显式delete
20     Summary& operator=(const this_type&) = delete;
21 private:

```

 复制代码


```
22     mutable lock_type    m_lock;           // 自旋锁
23     map_type             m_sales;          // 存储销售记录
24     。
```

Summary 类的职责是存储大量的销售记录，所以需要选择恰当的容器。

考虑到销售记录不仅要存储，还有对数据的排序要求，所以我选择了可以在插入时自动排序的有序容器 map。

不过要注意，这里我没有定制比较函数，所以默认是按照书号来排序的，不符合按销售量排序的要求。

（如果要按销售量排序的话就比较麻烦，因为不能用随时变化的销量作为 Key，而标准库里又没有多索引容器，所以，你可以试着把它改成 unordered_map，然后再用 vector 暂存来排序）。

为了能够在多线程里正确访问，Summary 使用自旋锁来保护核心数据，在对容器进行任何操作前都要获取锁。锁不影响类的状态，所以要用 mutable 修饰。

因为有了 RAII 的 SpinLockGuard（第 21 讲），所以自旋锁用起来很优雅，直接构造一个变量就行，不用担心异常安全的问题。你可以看一下成员函数 add_sales() 的代码，里面还用到了容器的查找算法。

 复制代码

```
1 public:
2     void add_sales(const sales_type& s)      // 非const
3     {
4         lock_guard_type guard(m_lock);       // 自动锁定，自动解锁
5
6         const auto& id = s.id();             // const auto自动类型推导
7
8         if (m_sales.find(id) == m_sales.end()) { // 查找算法
9             m_sales[id] = s;                 // 没找到就添加元素
10            return;
11        }
12
13        m_sales[id].inc_sold(s.sold());       // 找到就修改销售量
14        m_sales[id].inc_revenue(s.revenue());
15    }
```

Summary 类里还有一个特别的统计功能，计算所有图书销量的第一名和最后一名。这用到了 minmax_element 算法（[第 13 讲](#)）。又因为比较规则是销量，而不是 ID 号，所以还要用 lambda 表达式自定义比较函数：

 复制代码

```
1 public:
2     minmax_sales_type minmax_sales() const    //常函数
3     {
4         lock_guard_type guard(m_lock);        // 自动锁定，自动解锁
5
6         if (m_sales.empty()) {                // 容器空则不处理
7             return minmax_sales_type();
8         }
9
10        auto ret = std::minmax_element(        // 求最大最小值
11            std::begin(m_sales), std::end(m_sales), // 全局函数获取迭代器
12            [](const auto& a, const auto& b)      // 匿名lambda表达式
13            {
14                return a.second.sold() < b.second.sold();
15            });
16
17        auto min_pos = ret.first;              // 返回的是两个迭代器位置
18        auto max_pos = ret.second;
19
20        return {min_pos->second.id(), max_pos->second.id()};
21    }
```

服务端主线程

好了，所有的功能类都开发完了，现在就可以把它们都组合起来了。

因为客户端程序比较简单，只是序列化，再用 ZMQ 发送，所以我就不讲了，你可以课下去看 [GitHub](#) 上的源码，今天我主要讲服务器端。

在 main() 函数开头，首先要加载配置文件，然后是数据存储类 Summary，再定义一个用来计数的原子变量 count（[第 14 讲](#)），这些就是程序运行的全部环境数据：

 复制代码

```
1 Config conf;                // 封装读取Lua配置文件
2 conf.load("./conf.lua");    // 解析配置文件
3
4 Summary sum;                // 数据存储和统计
```

接下来的服务器主循环，我使用了 lambda 表达式，引用捕获上面的那些变量：

[复制代码](#)

```
1 auto recv_cycle = [&]()           // 主循环lambda表达式
2 {
3     ...
4 };
```

主要的业务逻辑其实很简单，就是 ZMQ 接收数据，然后 MessagePack 反序列化，存储数据。

不过为了避免阻塞、充分利用多线程，我在收到数据后，就把它包装进智能指针，再扔到另外一个线程里去处理了。这样主循环就只接收数据，不会因为反序列化、插入、排序等大计算量的工作而阻塞。

我在代码里加上了详细的注释，你一定要仔细看、认真理解：

[复制代码](#)

```
1 auto recv_cycle = [&]()           // 主循环lambda表达式
2 {
3     using zmq_ctx = ZmqContext<1>;           // ZMQ的类型别名
4
5     auto sock = zmq_ctx::recv_sock();        // 自动类型推导获得接收Socket
6
7     sock.bind(                               // 绑定ZMQ接收端口
8         conf.get<string>("config.zmq_ipc_addr")); // 读取Lua配置文件
9
10    for(;;) {                                // 服务器无限循环
11        auto msg_ptr =                       // 自动类型推导获得智能指针
12            std::make_shared<zmq_message_type>();
13
14        sock.recv(msg_ptr.get());            // ZMQ阻塞接收数据
15
16        ++count;                             // 增加原子计数
17
18        std::thread(                          // 再启动一个线程反序列化存储，没有用async
19            [&sum, msg_ptr]()                // 显式捕获，注意！！
20            {
21                SalesData book;
22            });
```



```

23         auto obj = msgpack::unpack(          // 反序列化
24             msg_ptr->data<char>(), msg_ptr->size()).get();
25         obj.convert(book);
26
27         sum.add_sales(book);                  // 存储数据
28     }).detach();                             // 分离线程，异步运行
29 }                                              // for(;;)结束
30

```

你要特别注意 lambda 表达式与智能指针的配合方式，要用值捕获而不能是引用捕获，否则，在线程运行的时候，智能指针可能会因为离开作用域而被销毁，引用失效，导致无法预知的错误。

有了这个 lambda，现在就可以用 `async`（[🔗第 14 讲](#)）来启动服务循环：

```

1 auto fu1 = std::async(std::launch::async, recv_cycle);
2 fu1.wait();

```

 复制代码

现在我们就能够接收客户端发过来的数据，开始统计了。

数据外发线程

`recv_cycle` 是接收前端发来的数据，我们还需要一个线程把统计数据外发出去。同样，我实现一个 lambda 表达式：`log_cycle`。

它采用了 HTTP 协议，把数据打包成 JSON，发送到后台的某个 RESTful 服务器。

搭建符合要求的 Web 服务不是件小事，所以这里为了方便测试，我联动了一下《透视 HTTP 协议》，用那里的 OpenResty 写了个的 HTTP 接口：接收 POST 数据，然后打印到日志里，你可以参考[🔗第 41 讲](#)在 Linux 上搭建这个后台服务。


`log_cycle` 其实就是一个简单的 HTTP 客户端，所以代码的处理逻辑比较好理解，要注意的知识点主要有三个，都是前面讲过的：

读取 Lua 配置中的 HTTP 服务器地址和周期运行时间（[🔗第 17 讲](#)）；

JSON 序列化数据 (🔗第 15 讲) ;


HTTP 客户端发送请求 (🔗第 16 讲) 。

你如果有点忘了，可以回顾一下，再结合下面的代码来理解、学习：

 复制代码

```
1  auto log_cycle = [&]()                // 外发循环lambda表达式
2  {
3      // 获取Lua配置文件里的配置项
4      auto http_addr = conf.get<string>("config.http_addr");
5      auto time_interval = conf.get<int>("config.time_interval");
6
7      for(;;) {                        // 无限循环
8          std::this_thread::sleep_for(time_interval * 1s); // 线程睡眠等待
9
10         json_t j;                    // JSON序列化数据
11         j["count"] = static_cast<int>(count);
12         j["minmax"] = sum.minmax_sales();
13
14         auto res = cpr::Post(         // 发送HTTP POST请求
15             cpr::Url{http_addr},
16             cpr::Body{j.dump()},
17             cpr::Timeout{200ms}      // 设置超时时间
18         );
19
20         if (res.status_code != 200) { // 检查返回的状态码
21             cerr << "http post failed" << endl;
22         }
23     }                                // for(;;)
24 };                                  // log_cycle lambda
```

然后，还是要在主线程里用 `async()` 函数来启动这个 `lambda` 表达式，让它在后台定时上报数据。

 复制代码

```
1  auto fu2 = std::async(std::launch::async, log_cycle);
```

这样，整个书店程序就全部完成了，试着去编译运行一下看看吧。

小结

好了，今天我就把书店示例程序从头到尾给讲完了。可以看到，代码里面应用了很多我们之前讲的 C++ 特性，这些特性互相重叠、嵌套，紧凑地集成在了这个不是很大的程序里，代码整齐，逻辑清楚，很容易就实现了多线程、高性能的服务端程序，开发效率和运行效率都非常高。

我再对今天代码里的要点做个简单的小结：

1. 编写类的时候要用好 final、default、using、const 等关键字，从代码细节着手提高效率和安全性；
2. 对于中小型项目，序列化格式可以选择小巧高效的 MessagePack；
3. 在存储数据时，应当选择恰当的容器，有序容器在插入元素时会自动排序，但注意排序的依据只能是 Key；
4. 在使用 lambda 表达式的时候，要特别注意捕获变量的生命周期，如果是在线程里异步执行，应当尽量用智能指针的值捕获，虽然有点麻烦，但比较安全。

那么，这些代码是否对你的工作有一些启迪呢？你是否能够把这些知识点成功地应用到实际项目里呢？希望你能多学习我在课程里给你分享的开发技巧和经验建议，熟练地掌握它们，写出媲美甚至超越示例代码的 C++ 程序。

课下作业

最后是课下作业时间，这次就不是思考题，全是动手题，是时候检验你的编码实战能力了：

1. 添加 try-catch，处理可能发生的异常（[🔗第 9 讲](#)）；
2. 写一个动态库，用 Lua/Python 调用 C++ 发送请求，以脚本的方式简化客户端测试（[🔗第 17 讲](#)）；
3. 把前端与服务器的数据交换格式改成 JSON 或者 ProtoBuf（[🔗第 15 讲](#)），然后用工厂类封装序列化和反序列化功能，隔离接口（[🔗第 19 讲](#)、[🔗第 20 讲](#)）。

再补充一点，在动手实践的过程中，你还可以顺便练习一下 Git 的版本管理：不要直接在 master 分支上开发，而是开几个不同的 feature 分支，测试完确认没有问题后，再合并到主干上。

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友。我们下节课见。

课外小贴士

1. Boost程序库里有一个multi_index库，提供多索引容器，能够以不同的接口访问容器里的元素，比如同时按ID号和销售金额排序。
2. 为了写起来“省事”，代码中使用了泛型的lambda表达式，所以只能用C++14标准来编译。
3. 后端RESTful服务器的源码在GitHub项目 [http_study的www/lua/cpp_study.lua](http://study.www.lua/cpp_study.lua)，如果你感兴趣的话可以去看看。

618 课程特惠

618 好课 5 折起

优惠口令立减 ¥15

618gogogo



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 知识串讲（上）：带你开发一个书店应用

下一篇 期末测试 | 这些C++核心知识，你都掌握了吗？

精选留言 (7)

写留言

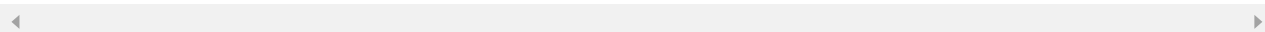


robonix

2020-06-28

老师，还想问一下，为啥不用std::lock_guard，自己写一个lock呢，只为了性能嘛？

作者回复: 示例代码，当然都是自己写出来比较好了，可以实践一下编码准则。



1



robonix

2020-06-28

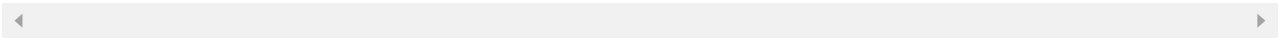
代码里显式声明了转移构造和转移赋值函数，这样，在放入容器的时候就避免了拷贝，能提高运行效率。

那么被转移的类会被掏空了，使得内部数据无效吗？

展开 ∨

作者回复: 需要理解转移语义，它的目的就是要将原对象的内容给“偷走”，转移到新的对象里。

这样原对象就空了，但数据依然是有效的，比如0、nullptr，只是没有了实际意义，可以被安全、轻量地销毁。



1



有学识的兔子

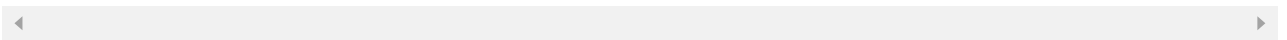
2020-06-26

我碰到一个pybind11的问题：代码Client.cpp使用了第三方zmq组件，如果要转化成python可以调用的模块，除了适配Client.cpp自身接口需要用pybind11声明外，zmq涉及到的接口也要做么？

看转换的格式比较固定，是不是有自动化的工具来做这件事呢？

展开 ∨

作者回复: 可以把zmq的调用封装起来，不对外暴露zmq接口，Python调用只传递几个参数。



有学识的兔子

2020-06-26

- 1、Thread生成的地方，没有去做异常检查，我不太确定需不需要？
- 2、假如使用python脚本去简化客户端测试，是不是通过PYBIND11的方式把Client.cpp里的接口转化成python能够加载模块，在利用python测试该模块？
- 3.可以将SalesData里面涉及pack和unpack的部分拆分出来，用工厂方法进行替换；工厂类可以借用STL将不同类型数据格式和对应工厂类映射起来；在通过配置文件增加该类型的...

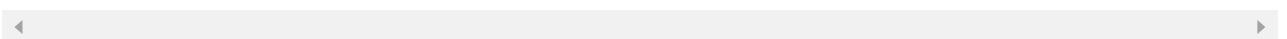
展开 ∨

作者回复:

1.只要没有显式声明noexcept的地方，其实都应该加上try-catch。

2.对，用C++写底层接口，然后用Python、lua去调用。

3.思路很对。





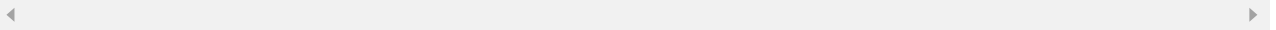
沉淀的梦想

2020-06-25

为什么不用智能指针 `unique_ptr`，而是一定要自己重新写一个 `SpinLockGuard`？

作者回复: `unique_ptr`只能管理对象的生命周期，自动销毁堆上的对象。而`SpinLockGuard`的目的是在生命周期结束时自动解锁。

虽然用的都是RAII技术，但两者的行为、作用不同。



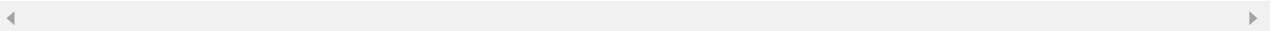
LDxy

2020-06-25

课下作业第3应该说客户端和服务端吧

展开 ∨

作者回复: 可以看一下21讲，这个服务器前面是前端服务，不是直接的客户端，当然，说是客户端也可以，毕竟是示例程序，不那么严格。



木瓜777

2020-06-25

每次接受请求，都开启一个线程，是否合理？

展开 ∨

作者回复: 每个请求开新线程的代价是比较高的，但课程里的代码只是为了演示目的，实际项目里最好用线程池。

